Efficient Procedural Generation of Forests

Julian Kenwood
University of Cape Town
nomad010@gmail.com

James Gain
University of Cape Town
jgain@cs.uct.ac.za

Patrick Marais
University of Cape Town
patrick@cs.uct.ac.za

ABSTRACT

Forested landscapes are an important component of many large virtual environments in games and film. In order to reduce modelling time, procedural methods are often used. Unfortunately, procedural tree generation tends to be slow and resource-intensive for large forests.

The main contribution of this paper is the development of an efficient procedural generation system for the creation of large forests. Our system uses L-systems, a grammar-based procedural technique, to generate each tree. We algorithmically modify L-system tree grammars to intelligently use an instance cache for tree branches. Our instancing approach not only makes efficient use of memory but also reduces the visual repetition artifacts which can arise due to the granularity of the instances. Instances can represent a range of structures, from a single branch to multiple branches or even an entire tree.

Our system improves the speed and memory requirements for forest generation by 3–4 orders of magnitude over naïve methods: we generate over $1,000,000$ trees in 4.5 seconds, while using only 350MB of memory.

Keywords

procedural tree generation, L-systems, instancing

1 INTRODUCTION

When large forests are used in CGI they are often created using procedural methods, due to their inherent geometric complexity. Unfortunately, the memory requirements of a procedural approach can be prohibitive. For example, some tree generation methods require as much as 10MB per tree [1]. Using such schemes, even a relatively small forest of $1,000$ trees would require much more memory than most commodity computer systems support. In addition to large memory requirements, procedurally creating a large forest from scratch could take minutes or even hours. Forests are thus usually created in an off-line process, which limits their use in games and interactive media.

We explore the problem of procedurally generating complete forests, focussing on algorithms and optimisations that facilitate the creation of very large forests, in the range of $10,000$ trees or more, in a few seconds.

We propose a new system for generating large numbers of trees with a fixed memory budget. We use L-systems to generate trees and introduce an optimisation to the L-system grammar that enables efficient caching of tree sub-branches. This allows the creation of subsequent trees to be accelerated, whilst also saving memory.

The focus of our work is not on the rendering of realistic trees, but rather on the often expensive procedural methods that underpin such systems. Consequently we illustrate our optimizations on basic branching tree structures and make no use of billboards, complex texturing and so on.

Our optimisations allow the generation of very large forests in a few seconds and with low memory overhead. This work is applicable to a broad range of L-systems and can thus supplement systems which currently make use of such a procedural approach.

The remainder of this paper is laid out as follows. Section 2 presents relevant background. Basic L-system formalism is introduced in Section 3. The optimisations that we have developed are presented in Section 4 and Section 5. The creation of tree geometry is discussed in Section 6. Section 7 presents our results and discusses the performance of the system. Finally, Section 8 summarises our findings and contributions and provides suggestions for future work.

2 RELATED WORK

EcoSys [1] represents one of the earliest and best-known procedural tree generation systems. EcoSys is able to generate realistic looking forests, including plants and other foliage, from a relatively small amount of input, such as a heightmap of the landscape. The system allows for interactive editing of the parameters with built-in editors. Each individual plant is procedurally created using an L-system. L-systems provide a

set of match-replace rules that specify the appearance of the tree. These rules control everything from the texture and colour of the tree and its leaves to how the tree branches and how branches lengthen. L-systems allow modellers to present a set of rules that describe a particular species of tree.

Unfortunately, the amount of data generated for a single tree in EcoSys can be as much as 10MB. A small forest of $1,000$ trees results in a total of 10GB of data which cannot be rendered efficiently using current technology.

In order to reduce the amount of memory required the system uses *instancing* to generate a single tree that can be used in multiple places, which saves memory but can reduce realism. EcoSys only uses instances in cases where the resulting trees are likely to be similar.

EcoSys is able to render forests interactively using points and lines, but cannot achieve high enough visual quality and frame rates for games, even when executed on modern hardware [2].

The trees that are presented in modern games are usually made with a third-party library called *SpeedTree*[1]. SpeedTree, however, is proprietary software and companies have to pay a license fee to use it. SpeedTree generates trees using an offline process: either procedural generation or manual generation by an artist.

As with EcoSys, large forests are accommodated with instancing. The trees in the forest originate from a considerably smaller 'hero' tree set. Unfortunately, since the library of trees that an artist works with can be small, the same tree could be repeated in an unrealistic fashion, particularly when the game is intended for a very resource-limited platform. Recently, however, SpeedTree has added a WorldBuilder module which is able to export tree positions that exhibit fewer jarring repetition artifacts.

In this paper, we explore an alternative method for creating large forests. Similarly to EcoSys and SpeedTree, we use procedural methods to generate the individual trees in the forest. Unlike EcoSys and SpeedTree, we aim to use instancing to reduce the size of the forest *without* resorting to instancing entire trees. Our primary aim is to decrease the memory requirements for trees in forests without sacrificing visual quality.

3 L-SYSTEMS

Lindenmayer Systems, or L-systems, are used extensively in Procedural Graphics [5]. The rules for these systems are capable of describing complex structures such as plants [6] and buildings yet are simple enough to be created by modellers [4]. The simplest type of L-system — deterministic context-free L-systems (also called DOL-systems [7]) are simple match-replace rules that occur over a string of symbols. Each symbol has a specific meaning used later in the tree creation process. For instance, the symbol 'F' means to draw a cylinder at the current position, while a '+' symbol changes the orientation of the next cylinder.

$$\underbrace{O}_{\textit{Strict predecessor}} \rightarrow \underbrace{F\,F\,F\,F\,F}_{\textit{Derivation}}$$

The strict predecessor is a single symbol that should be transformed into a (possibly empty) sequence of symbols called the derivation. All rules in the L-system are applied simultaneously to the entire string of symbols. The number of times the rules are applied is called the generation of the string, corresponding to the required age of the output tree. The initial string of symbols, also called the axiom, is denoted by generation 0.

The symbols from the final string are used as drawing instructions for a turtle-like graphics module called the interpreter. Symbols with no meaning are simply discarded.

There are several drawbacks to DOL-systems in the context of tree creation. Most importantly, the output for a given generation of an L-system is always the same. This means that the only way to add variation is to create scaled copies of each tree type. This limitation can be addressed by using stochastic L-systems [3], which allow *multiple* derivations with associated probabilities that indicate the likelihood of selection.

A second drawback to DOL-systems is the difficulty of growing branches of a desired length: each tree is made of individual cylinders of equal length. This can be solved by introducing parameters: each symbol in the string can have additional parameters, which can exactly model the desired length, width, and other attributes.

Researchers have also found it useful to modify the DOL-system to add two symbols, [and], to assist in creating trees efficiently by controlling a stack of saved turtles; pushing and popping onto the stack, respectively.

Before moving on, we need to introduce the concept of a *module*. A module is a special symbol in the L-system that can used to achieve higher order functionality: it effectively represents a callout to a 'subroutine'. Modules can be distinguished from regular L-system symbols by the inclusion of parentheses, which surround 0 or more parameters. Our modifications make extensive use of modules.

4 BRANCH OPTIMISATION

Branching in trees is a crucial aspect of realistic growth. Unfortunately, L-system branches generate a significant amount of geometry. This problem is exacerbated

[1] http://www.speedtree.com

when creating large numbers of trees. Our system, like EcoSys and SpeedTree, attempts to solve this problem through the application of *instancing*. However, unlike these methods, we choose to perform instancing at the granularity of branches. Such fine-grained instancing allows branches to be shared across multiple trees, whilst saving memory and keeping some degree of visual differentiation.

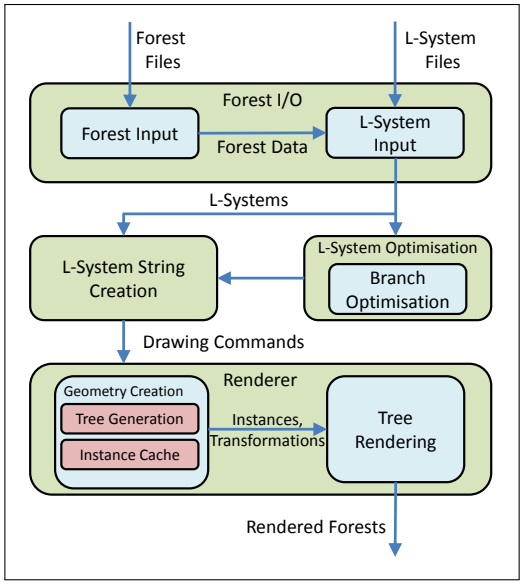

Figure 1: System structure - input L-systems are taken in together with forest generation parameters. The L-systems are then optimised to make use of instancing and geometry is generated to represent the various instanced structures. This information is then passed off to the renderer, along with positioning.

In the context of tree L-systems, branches represent additional *recursive* work that must be performed. With the help of instancing, we essentially memoise the result of this work so it can be re-used later. The basic architecture of our system is presented in Figure 1.

We use the stochastic nature of L-systems to decide which branches should be instanced and which created. A fixed *instancing probability*, P, is used to control instancing. This is a percentage represented by an *integral* number between 0 and 100, where 0 indicates no instancing and 100 indicates full instancing. We modify the L-system rules to reflect this probability. However, rules that are responsible for branching must first be detected.

4.1 Rule Detection

Each rule is examined to determine if it contributes branches to the tree. It is difficult to determine exactly which rules branch, so a heuristic is used instead. The bracket symbols, [and], are a common indicator of branching because they isolate state changes. The left bracket saves state information, such as position and orientation, which is restored at the right bracket. While this behaviour is useful in branching it is also applied to create leaves, as indicated in Table 1, as well as other non-branching phenomena.

$$L \mapsto [\,\hat{}\,\hat{}\; - f + f + f - \,|\, - f + f + f\,]$$

Table 1: A rule from an L-system used to draw a leaf. Brackets are employed but no branching occurs.

The heuristic uses brackets as an indicator of branching, but to filter out erroneous cases a further restriction is applied: the brackets must contain at least one *non-terminal* symbol. Non-terminals are the strict predecessors on the left-hand side of each rule. While this may still incorrectly identify rules as branching, it is significantly more accurate than using brackets alone.

$$X \mapsto [\,L\,]$$
$$L \mapsto \hat{}\,\hat{}\; - f + f + f - \,|\, - f + f + f$$

Table 2: A modified version of the L-system in Table 1. The second rule is incorrectly identified as a branching rule by our heuristic.

Segments of the rule symbols are identified as start and end points for the branch. The brackets and the symbols between them are tagged with an identifier. Branches represented by identical symbols share an identifier, the assumption being that the resultant geometry is the same. The identifiers are global in that they may be shared across different rules; they are used later to access one of several instance caches.

Algorithm 1 shows the detection process. In the pseudocode, `seenBranch` and `branchGUID` return information about the branch currently being examined. `seenBranch` returns true if it is identical to a previously seen branch. `branchGUID` returns the unique identifier of a previously seen branch. The function `addBranch` adds a branch to the global list of previously seen branches and returns its new unique identifier. `createBranch` creates a structure that packages the unique identifier and the branch symbol information for later use in the program.

4.2 Rule Modification

The rule modification process is complicated by the existence of stochastic rules.

We begin by looking at the simpler case of modification for deterministic L-systems. For such systems, each rule's right-hand side is replaced by several right-hand sides (meaning that the resulting rule becomes stochastic) depending on the number of branches that occur. If B branches are present on the right-hand side, then 2^B

```
detectBranching(rhs, nonTerminals)
output = []
for s = 1 → length(rhs)
  if rhs[s] = [ then
    t = s + 1
    for t → length(rhs)
      if rhs[t] = ] then break
    if t = length(rhs) then continue
    for u = s + 1 → t
      if rhs[u] in nonTerminals then
        continue s
    guid = -1
    if seenBranch(s, t) then
      guid = branchGUID(s, t)
    else seenBranch(s, t) then
      guid = addBranch(s, t)
    output += [createBranch(guid, s, t)]

return output
```

Algorithm 1: Branching RHS Detection.

$$
\begin{array}{ll}
 & \mapsto F\ A \\
A & \mapsto \overbrace{[\ \&\ F\ L\ !\ A\ ]}^{\text{Branch 1}}\ /\ /\ /\ /\ / \\
 & \quad \overbrace{[\ \hat{}\ F\ L\ !\ A\ ]}^{\text{Branch 2}}\ /\ /\ /\ /\ /\ /\ /\ \overbrace{[\ \&\ F\ L\ !\ A\ ]}^{\text{Branch 1}} \\
F & \mapsto F\ F \\
L & \mapsto [\hat{}\ \hat{}\ \{\ -\ f\ +\ f\ +\ f\ -\ |\ - \\
 & \quad f\ +\ f\ +\ f\ \}\ ]
\end{array}
$$

Table 3: A leafy tree L-system [5]. The A-rule has been annotated with information that marks the segments that branch. Each branch is tagged with a number that identifies the branch segment. Note that the third branch has the same identifier as the first branch due to having identical symbols.

new right-hand sides are created, representing the possibility of either instancing each branch or not.

If a branch is to be instanced, the relevant symbols are replaced by a *getInstance* module. Otherwise, other control modules, *startInstance* and *stopInstance* are inserted instead. These two modules demarcate segments of a string that correspond to branch information. Each takes two parameters: an *identifier* and an *age*. The identifier is the same as the one associated with the branch in the rule detection phase. The age is determined from the *getGeneration* function, which returns the generation at which the module was created.

Each right-hand side is given a probability, p, based on the instancing probability and the number of branches being instanced, calculated as follows:

$$p(I) = P^I \times (1-P)^{B-I}$$

where P is the probability of replacing a branch with an instance, B is the total number of branches and I is an index variable. For each right-hand side, the index variable is the number of times that the decision is made to instance a particular branch.

A simple binary number counting algorithm is used to enumerate these rules that is both efficient and easy to implement. It is only suitable if the number of branches in a rule is less than the bit-length of a machine's word size. In practice this constraint is not at all problematic. The disadvantages of this optimisation are evident from inspecting the output: the number of right-hand sides has greatly increased and each is significantly less humanly readable.

Stochastic L-systems add complexity in that the several right-hand sides may create branches. The above algorithm is performed on each original right-hand that contains branching segments. The relative probabilities of each group of newly created right-hand sides must reflect the original distribution. To enforce this, the equation for p is modified:

$$p(I) = P_{original} \times P^I \times (1-P)^{B-I}$$

where $P_{original}$ is the probability of the originating rule. Multiplying by the original probability ensures that the probabilities have the correct distribution.

The time required to apply this optimisation to a set of rules depends on the number of branches, B_i, and the length, L_i, of each right-hand side. The total computation and memory cost is bounded by $O(\sum 2^{B_i} L_i)$. The main source of the inefficiency of this algorithm is the number of right-hand sides that are created. In our investigations, B_i is rarely larger than three and is thus not problematic. It may also be possible to achieve the same effects using fewer right-hand sides. This is, however, left as future work.

The effectiveness of the branching optimisation depends on the instancing probability. If this is set too high, repetition will become evident. Conversely, setting it too low results in the memory required for a forest becoming too large. We use an instancing probability that varies as more trees are created to support instancing that becomes more aggressive as memory becomes scarcer.

The *getInstance*, *startInstance* and *stopInstance* symbols are used during interpretation to communicate with the higher levels of the system. *getInstance* indicates that the system should record the current position and orientation where an instance should be used to complete the tree. *startInstance* notifies the system that

all the geometry created until the corresponding *stopInstance* symbol forms a coherent instance usable as part of other trees. The instance cache is the device used to store and retrieve instances and is introduced next.

5 INSTANCE CACHE

The end result of the interpretation phase is geometry in the form of vertex and index buffers, required for rendering. Using the symbols introduced for the L-system rule modification process, we annotate the index buffers as they are created. In this way, we can determine which ranges of the index buffers correspond to branches with different sizes and properties. Multiple instances can exist within the same index buffer. For example, a tree branch could have an annotated sub-branch. For this reason, in addition to pointers to the vertex and index buffers, two integers are stored to denote the range in the buffers that correspond to the current instance.

Each index buffer range also stores metadata about the range. The age, species type and branch identification information are stored in hash tables to allow for easy and efficient access. The hash table data contains arrays of pointers to these ranges, which allows for efficient random selection. The Instance Cache can thus retrieve a random index buffer range based on any set of age and species criteria.

In addition to storing the buffer range, the transformation of geometry is retained. Each transformation is a matrix that represents the spatial orientation of the geometry in the index buffer. This information is necessary to correctly place the branch on a renderable tree. Finally, the orientation and positions of any *getInstance* modules that occurred in the branch string are recorded. These *getInstance* modules are used to indicate exit points for the instance which must be filled with other instances in order to generate a complete tree.

6 TREES FROM INSTANCES

Trees are created in the system in two distinct phases: *Hero Creation* and *Tree Placement*. Hero Creation runs the L-systems in order to fill up the various instance caches that exist. We use one instance cache per unique identifier generated in the rule modification phase. These heroes serve as the template geometry for tree and branch instances. Although each hero created is a correctly derived tree, they are not used directly for rendering purposes: the creation of trees for rendering is left to the Tree Placement phase.

The cumulative size of all geometry buffers is limited in our system depending on the available memory. For example, if one were creating a virtual environment where forests are not important, the maximum size could be very low, perhaps as little as 16MB. On the other hand, for environments where forests are a prominent feature, one can devote upwards of 256MB to the required buffer. The ability to define a range of cache sizes allows developers to tightly control the resources used to generate a forest.

As more hero trees are created, the memory space that can be devoted to geometry decreases. In order to allow for the continued creation of new geometry, we *increase* the instancing probability for each hero tree that is created. A higher instancing probability means that more *getInstance* symbols will occur in the L-system strings. In other words, fewer new branches are created and more instances are used.

The final stage in forest creation is Tree Placement. Tree Placement creates new trees by cutting and joining parts of the hero trees together and calculating the necessary transformations to place the trees on the terrain.

A renderable tree is represented by a collection of pointers to geometry buffer ranges that are stored in the Instance Cache. Creating a tree from the instance cache is done recursively. The algorithm takes the species of the desired tree, its generation and a transformation matrix describing the desired position and orientation, as input. Given this information, an instance is selected from the instance cache to serve as the base of the tree.

An instance is not limited to representing a single generation. Instead, it can represent the entire tree, a single generation or, more likely, several generations with exit points that need to be filled with sub-branches. The exit points describe not only the desired position and orientation relative to the start of the instance but also the desired age and branch identification of the instance that should fill the gap.

The algorithm recurses for each exit point required by the current instance. The age and branch information parameters are used to constrain the subsequent search of the instance cache. The instances are re-oriented by computing a placement transformation matrix. Given the desired orientation matrix, D, and the orientation matrix of the instance within its hero tree, M, we calculate the transformation to reorient an instance, T, as:

$$T = D \times M^{-1}$$

The initial desired orientation is passed as a parameter to the recursive function so it must be updated before it is passed into the next function call. To update the orientation we use the following formula:

$$D_{new} = D_{old} \times E$$

where D_{new} is the new orientation, D_{old} is D from above, and E is the orientation of the next exit point in relation to its hero tree.

Algorithm 2 shows the tree creation process described above. The recursive function, `CREATE`, takes several

```
create(output, direction, age, cache)
instance = cache.getInstance(age)
T = direction × instance.direction.inverse
output.addTreeInstance(instance.buffers, T)
for i = 1 → length(instance.exitPoints)
    newAge = instance.exitPoints[i].exitAge
    exit = instance.exitPoints[i].direction
    newDirection = direction × exit
    create(output, newDirection, newAge, cache)
```

Algorithm 2: Renderable tree building process.

input arguments that describe the instance we wish to find. In the algorithm listed above, we only search the instance cache by generation; in practice we use other criteria as well. The desired orientation, `direction`, is used to compute the transformation, T, needed to correctly draw the geometry buffer. The `output` contains a list of geometry buffers that we should draw and their correcting transformations. The `addTreeInstance` method simply appends the buffers and transformations to this list. The recursive function terminates when all exit points have been filled. Although this function is recursive, it remains significantly faster than regular L-system creation as no geometry is created. Only geometry in the instance caches are used.

Rather than applying the transformations to the geometry data immediately, the transformation is stored so that the renderer can perform the transformation on the fly. This allows the geometry data to be efficiently reused across multiple trees and branches. The index buffer range and the required transformation are saved to the tree object for use with the renderer. The end result of this process is a collection of pointers to index buffer ranges and the transformations necessary to correctly render the tree at a particular position.

7 RESULTS

Testing was done on an Intel Core i5 2.80GHz quad-core machine with 8GB of RAM and an NVidia 580GTX graphics card. To avoid interfering memory requests from other applications or processes, we limited the system to using 4GB of RAM.

We split our tests into two groups. For the first group, we kept the forest size constant, while testing the run-time of different cache schemes. The forest size was kept at 10,000 trees and the following cache sizes were tested: 16MB, 32MB, 64MB, 128MB, 256MB, and 512MB. The second group of tests kept the cache size at a constant 128MB and created forests of up to 1,000,000 trees, and measured both the memory and run-time of the forest creation process.

Figure 2 shows the results of creating forests with varying cache sizes. In most cases, the majority of the time is spent creating hero trees to fill the various caches.

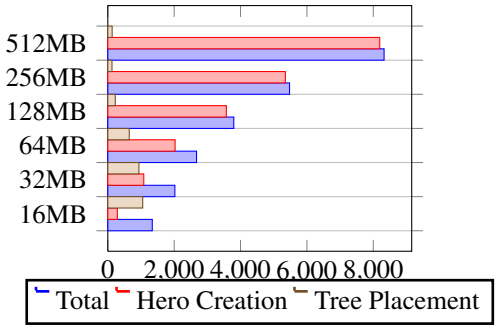

Figure 2: Time (in milliseconds) required to create 10,000 trees, as a function of cache size. Hero tree creation accounts for the largest proportion of processing time for larger cache sizes.

From our results, hero creation time grows roughly linearly with increasing cache size. The 16MB cache takes approxmately one second to fill, while the 512MB cache takes up to eight seconds. The hero creation process requires, on average, 25 milliseconds per MB of cache.

By comparison, tree placement generally requires much less computation time, although it does scale with the number of trees being created. However, for smaller cache sizes, tree placement requires a larger proportion of total running time for a fixed number of trees. There are two reasons for this. First, a smaller cache can be filled significantly faster than a large cache. Second, each additional hero tree depletes the percentage of cache space available much more rapidly for small cache sizes. This has a knock-on effect on the instancing probability used to generate new hero tree. A hero tree created with a high instancing probability is likely to contain many instance exit points (*getInstance* symbols in the L-system string). Consequently, Algorithm 2 will require many more recursive calls on average and, thus, take longer to run. This phenomenon is not seen in the large cache sizes since the instancing probability increases much more slowly.

As the cache size increases, the time required for tree placement reduces from one second, for the 32MB cache, to 130 milliseconds for the 512MB cache. This equates to a placement time of 0.1 milliseconds per tree and 0.013 milliseconds per tree, respectively. The total time to create and place 10,000 trees ranges from 1.3 seconds for the 16 MB cache to 8.4 seconds for the 512MB cache. The cache size is an important choice that must be made by the user of the system. If the desired output is a small forest, a small cache size should be chosen and vice versa. The disadvantage to choosing larger cache sizes, however, is the significantly longer time that users must wait in order for the cache to fill.

To determine the general utility of our instancing approach, we also evaluated the running time for forest creation *without* using any instancing. Unfortunately,

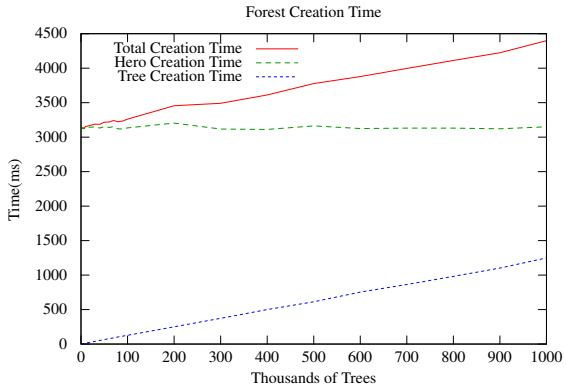

Figure 3: Time (in milliseconds) required to create forests for a cache size of 128MB. Hero tree creation accounts for most of the time, and is almost constant. Tree creation, which includes tree placement, increases linearly with the number of trees, but very slowly: creation of 1,000,000 trees requires only 1.5 seconds.

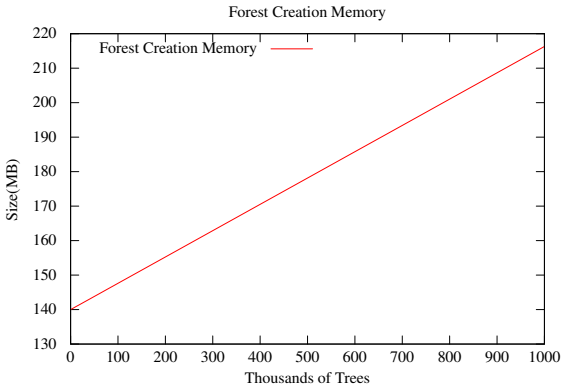

Figure 4: Memory (in megabytes) required to create a forest for a cache size of 128MB.The instance cache accounts for most of the memory usage. Each individual tree typically requires an average of only 100 bytes.

the system ran out of memory when attempting to create all $10,000$ trees. The largest size that we were able to create was approximately $2,000$ trees over sixty seconds. Based on our testing, we estimate that creating $10,000$ trees would require at least 300 seconds to complete. It is clear that our caching system is markedly faster than the uninstanced approach.

Our next set of tests show the running time and memory consumption of our method as the forest size increases.

As can be seen in Figure 3, the time required to add new renderable trees grows very slowly. Although this graph only shows results for a 128MB cache, the other cache sizes exhibit similar behaviour. Even for a million trees, the majority of time, about 3 seconds, is taken up with hero creation. A million trees only requires 1.5 seconds to create, which is equivalent to approximately 650 trees per millisecond. The time required to place trees — a component of the creation time — grows linearly with the desired number of trees. As noted above, our system performs better than the uninstanced approach: we can create a million trees in the time that the uninstanced method is only able to create three hundred trees.

Figure 4 shows the memory requirements for each additional tree in the forest. Memory usage grows approximately linearly. The memory usage metric is the sum of the instance cache size, the size of all renderable trees and the size of all textures in the trees. The graph shows that, even up to a million trees, the memory requirements are dominated by the cache size. The memory requirements grow very slowly with the number of trees in the forest, which highlights an important advantage over the uninstanced approach. Without instancing, each additional tree consumes a large amount of graphics memory, which can be severely limiting on all but the most recent hardware. In our approach, however, each additional tree consumes less than 100 bytes, on average.

The effect of instancing changes with cache size: for a small cache far more instancing takes place. Figure 5 shows a forest with 100,000 trees and a cache size of 32MB (left) and 128MB (right). The shade of red indicates the extent to which branch instances occurred in the forest, with grey indicating that branch/part of the tree was not instanced at all. Note that the shading does not indicate spatial proximity or branch indices, but simply the degree of instancing. It is readily apparent that the larger cache size dramatically reduces the amount of instancing. For the 128MB cache, no more than 4 instances of any branch were used, while for the 32 MB cache, no more than 27 instances of any branch structure were re-used throughout the forest.

In the next section, we will discuss some of the disadvantages of our approach, in particular, the visual artifacts that can occur when using random instancing.

7.1 Limitations

Using instancing for procedural forest generation is not without its drawbacks. An important criticism is the possible reduction in visual quality due to excessive reuse of tree geometry. We attempt to reduce the effect on appearance by letting the non-deterministic nature of the L-systems decided where instances should be placed. This can still lead to problems: the L-system could randomly decide two trees that are near to each other should use the same instances, or, even worse, be constructed entirely from the same instance.

Although identical trees are unavoidable due to the random nature of L-systems, certain steps can be taken to mitigate this effect. The first is to use L-systems which are markedly non-deterministic, in that they are able to produce a large number of varied trees. The second approach is to detect when we are about to place an of-

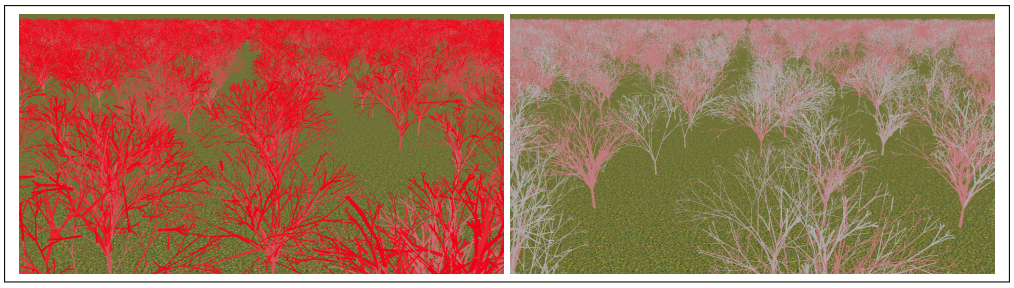

Figure 5: The effect of instancing for a forest of 100,000 trees. The shade of red indicates how frequently that branch structure was re-used in the forest. For the small cache (32MB, left) much more instancing is evident, as one would expect. However, no more than 27 instances were used for any branch structure. For the large cache (128MB, right) the number was only 4, as shown by the much lighter shading of red.

fending instance, one which is too close to another instance of the same type, and replace it with another instance. This second approach is made more difficult by the random nature of L-systems. The L-system could, for example, decide to add multiple instances to the cache which all represent the same geometry. In this case we would not even be aware that we are using the same geometry when using different instances.

Fortunately, the visual artifacts arising from instancing are not necessarily a problem. Informal user tests (a user-guided fly-through of the forest) revealed that most users do not notice that instances existed. Indeed, identical trees that are nearby often go unnoticed if oriented at a random angle and branches that are identical can appear at different levels of the tree which further masks their similarities.

8 CONCLUSIONS

We present a new scheme which significantly accelerates L-system tree creation and reduces memory overheads. By dynamically modifying the tree L-systems and making careful use of instancing, we can create large and varied forests quickly whilst using a bounded amount of memory. This is accomplished by filling a special fixed-size instance cache with sub-branch geometry derived from a much smaller set of 'hero' tree templates. Rendering is accomplished by looking up the appropriate geometry buffers in the instance cache and issuing draw calls using the associated transformation and texture metadata. Each new tree requires less than a 100 bytes of storage on average and takes less than 0.1 milliseconds to create. During our tests, we were able to create a forest of one million trees using approximately 350MB of memory in under 4.5 seconds. By contrast, the naïve algorithm was only able to generate 300 trees in the same amount of time.

There are several avenues for future work. Instancing improves memory requirements but may give rise to visually jarring repetition. In order to correct this behaviour we would need to scan the instance cache to detect when duplicate geometry is added so that we could ignore it. This is not an easy task since it requires complicated matching on the geometry and would likely slow the system down significantly. Alternatively, one could make the simplifying assumption that the same substring (the part of the string that represents the branch) represents the same geometry. Comparing strings instead of geometry is significantly easier (and more efficient).

Finally, while we have demonstrated our technique at work with L-systems, it is possible that other procedural tree generation algorithms could also benefit from our instance cache scheme.

9 ACKNOWLEDGEMENTS

This research was supported by an NRF/THRIP grant and the Centre for High Performance Computing.

10 REFERENCES

[1] Deussen, O.,Hanrahan, P., Lintermann, B., Měch, R., Pharr, M., and Prusinkiewicz, P. Realistic modeling and rendering of plant ecosystems, In: SIGGRAPH '98, p. 275-86, 1998.

[2] Deussen, O., Colditz, C., Stamminger, M., and Drettakis, G. Interactive visualization of complex plant ecosystems, In: Proceedings of the conference on Visualization, p. 219-26, 2002.

[3] Eichhorst, P., and Savitch, W. Growth functions of stochastic Lindenmayer systems, Information and Control, 45(3), p.217-28, 1980.

[4] Müller, P., Wonka, P., Haegler, S., Ulmer, A., and Van Gool, L. Procedural modeling of buildings, In: SIGGRAPH '06, p. 614-23, 2006.

[5] Prusinkiewicz, P., Lindenmayer, A., and Hanan, J. The algorithmic beauty of plants, The Virtual Laboratory, 1991.

[6] Prusinkiewicz, P. Graphical applications of L-systems, In: Proceedings of Graphics Interface, 1986.

[7] Salomaa, A., and Rozenberg, G. Handbook of Formal Languages, 1997.